Bounded delay for a free address

Hesselink, Wim H.

Published in:
Acta informatica

Publication date:
1996

Acta Informatica 33, 233–254 (1996)

Bounded delay for a free address

Wim H. Hesselink[1]

Department of Mathematics and Computing Science, Rijksuniversiteit Groningen, Postbox 800, 9700 AV Groningen, The Netherlands

Received February 2, 1994/January 24, 1995

Abstract. The problem is to let n processes concurrently and repeatedly search for free addresses in a range of m addresses. The search must be wait-free: a searching process finds an address in a bounded number of steps. Three solutions are presented. The first one has large atomic actions. The second one is only correct if $m \geqq (r+1) \cdot n$ where r is the maximum number of used addresses. The third solution is always partially correct. It is wait-free if $m > r + 2 \cdot n$. This solution has a worst-case waiting time quadratic in n and an amortized waiting time linear in n, even linear in the number of active processes.

1. Introduction

In concurrent programming with shared memory the following problem seems to have many applications. Assume that n processes concurrently use addresses from a set of m addresses (actually, thoughout this paper, the word address can be replaced by resource, number, name, etc.). Over time, a process may need a free address or an address may become available. The problem is to implement the search for a free address in a wait-free manner, i.e., without idle waiting primitives and such that any searching process P obtains a free address in a bounded number of steps of P, see [9]. Notice that wait-free implementations have an aspect of fault tolerance: if some processes stop executing, this does not affect the functioning of the other processes.

Since the bound on the number of steps may be large, e.g., of order quadratic in n, the term "wait-free" is somewhat misleading. We therefore prefer the term "bounded delay", which we shall use as a synonym.

Remark. The term "bounded delay" goes back (at least) to [8]. In that paper, however, bounded delay stands for what now would be called weak fairness. (End of remark)

[1] e-mail: wim@cs.rug.nl

1.1 Problem specification

We assume that used.k indicates whether address k is being used. So used is an array, as declared in

type *address* $= 0..m-1$;

var used: **array** *address* **of** *boolean*;

initially $(\forall k \in address :: \neg \text{used}.k)$.

In all such quantifications, dummy k ranges over the bounded type *address*.

From the point of view of the processes, used is a ghost variable. So, the processes cannot inspect used. They may modify array used only by setting used.i:= *true*. Moreover, this is only allowed under the precondition $\neg$used.i. More precisely, each process is to perform an unbounded repetition

while *true* **do** *search* **od**,

and each execution of *search* performs exactly once the command

$$U\text{: used}.E := \textit{true}.$$

Here E must be a private expression of the process, i.e., an expression that cannot be modified by other processes. Since command U is only allowed under precondition $\neg$used.E, we have the proof obligation that every process P always satisfies

(P0) $$P \textbf{ at } U \Rightarrow \neg \text{used}.E.$$

Bounded delay for command *search* means that there is a constant C such that every execution of *search* of any process P terminates within C actions of process P.

A process that gets an address is *not* kept responsible for this address. For example, it may broadcast the address to a set of other processes and it need not keep track of what happens to the address. In particular, the process that gets the address may ask for a new address before the first one can be released.

We therefore assume the existence of an environment that acts as a garbage collector. In other words, from time to time, this environment may choose an address k that satisfies used.k and make it free by performing the atomic action

$$\textit{Env}\text{: } \langle F; \text{used}.k := \textit{false} \rangle,$$

where F is some command that signals the availability of address k to the processes. So, used is not a ghost variable from the point of view of the environment.

Notice that we are dealing with an abstraction: we are not interested in the *usage* of the addresses or in the choice of k (i.e., the implementation of the garbage collector); from the point of view of searching processes, it only matters that some addresses become available again.

The number of addresses needed clearly depends on the maximal number of addresses that can be used simultaneously. We therefore introduce a constant r and

we assume that the environment preserves the invariant

(P1) $$(\#k :: \mathsf{used}.k) \leqq r,$$

where we write $(\#k :: A)$ to denote the number of values k that satisfy A. The upper bound r will be used to express conditions under which the algorithms are correct and have bounded delay.

1.2 Related problems in the literature

Our problem is related to the renaming problem considered in [4]. The main difference is that in the renaming problem a process that has obtained an address terminates, whereas in our problem the process gives the address to the environment and starts looking for a new address; the environment may be using the address for sometime before making it available again to the searching processes. Therefore, the maximum number of used addresses is a parameter, r, of our problem. This parameter may be bigger than n.

Our problem can be described as "repeated renaming", as mentioned briefly in [2]. We regard repeated renaming, however, as an inadequate abstraction. In fact, in the renaming problem, the address to be found is intended to serve as the new process name. In the case of repeated renaming, this leads to unnecessary complications when the process names are to be used in the program. Moreover, in our motivating example (cf. [9, 10]), the process that obtains the address may lose interest in the address long before other processes (represented here by the environment) do so. It is therefore misleading to treat the address obtained as a new name for the obtaining process.

As mentioned in [4, 2], the renaming problem becomes trivial (requires no communication) when the list of occurring process names is known to the processes. In our problem there is no reason to disallow the use of process names in the programs. Indeed, we do get simpler algorithms by using the process names to break the symmetry. One may say therefore that our problem and the renaming problem are disjoint up to triviality.

In spite of these differences, our algorithms can be compared with three algorithms in [4, 5], which indeed are wait-free. More precisely, the algorithm Unique-1 of [4] has quadratic time complexity and quadratic space complexity. Our algorithm *search2* below uses the same primitives and works in linear time and in space proportional to $r \cdot n$. The algorithm Unique-2 of [4] uses linear space, but has exponential time complexity. The algorithm in Figure 4 of [5] uses linear space and has cubic time complexity. Our algorithm *search3* below uses stronger primitives (read-and-set and consensus) and requires linear space and quadratic time (linear time in amortized sense).

The main differences, however, are that in our setting a process is allowed to ask for a new address long before its old addresses have been released, whereas the processes in [4, 5] ask for a new name only once, and on the other hand, that our processes have a known identity whereas the processes of [4, 5] are anonymous.

The paper [2] treats the renaming problem in a model with asynchronous message passing. Here no wait-free implementation is possible. Indeed, in the algorithms proposed, each process waits for $n - t - 1$ acknowledgements where

t is the maximum number of faulty processes. This may lead to unbounded waiting time.

1.3 Overview of the paper

The aim of our paper is to provide an implementation of *search* and an atomic command F, such that the combination satisfies the requirements specified in Sect. 1.1. An additional criterion is graceful degradation: it is desirable that the algorithm remains correct and possibly fair in overload situations where (P1) is violated, i.e., the number of used addresses is bigger than specified.

We give three solutions, two more or less trivial ones and a third one that is delicate. The first solution is presented in Sect. 2; it is fast but has a coarse grain of atomicity. It is correct if and only if $m \geqq r + n$. In Sect. 3, we set the stage for the other two solutions. The second solution, *search2* in Sect. 4, has a fine grain of atomicity but requires much address space, namely $m \geqq (r + 1) \cdot n$. Actually, it seems likely that wait-free solutions with this restricted repertoire to our problem require at least $r \cdot n$ addresses.

The third solution is *search3* in Sect. 5. This solution may be characterized as a concurrent circular search of co-operating processes. In Sect. 5, we show that it is partially correct and that $m \geqq 2 \cdot n - 1$ and a weak fairness assumption are sufficient to preclude individual starvation. So this solution has graceful degradation. In Sects. 6 and 7, we investigate its complexity under the assumption that $m > r + 2 \cdot n$. In Sect. 6, it is shown that *search3* is then wait-free. So this algorithm requires less memory space than *search2* . We also show that its worst-case time complexity is quadratic in n. In Sect. 7, we show that the amortized time complexity is linear in the number of active processes. In Sect. 8, a modification is proposed with better average behaviour, at least according to a small set of test simulations. These simulations also support the claim of graceful degradation. In Sect. 9, we give a comparison between the various solutions.

2 Using a stack and a coarse grain of atomicity

In this section, we present an efficient solution and we argue that its grain of atomicity may be too coarse. This solution uses a stack of free addresses, as implemented by

var x: **array** *address* **of** *address*

; t: *integer*;

initially $\mathrm{t} = m \wedge (\forall j, k \in \mathit{address}: j \neq k: \mathbf{x}.j \neq \mathbf{x}.k)$.

Conceptually, the stack consists of the values $\mathbf{x}.k$ with $0 \leqq k < t$. Initially, all addresses are on the stack. Using [and] to represent **begin** and **end**, we implement *search* by the compound command

[*pop*

; U: used i := *true*],

where i is a private variable and where *pop* is the atomic command

$$pop\colon \langle \mathbf{t} := \mathbf{t} - 1;\ i := \mathbf{x}.\mathbf{t} \rangle.$$

Atomic command F of the environment is implemented by

$$push\colon \langle \mathbf{x}.\mathbf{t} := k;\ \mathbf{t} := \mathbf{t} + 1 \rangle.$$

This solution is correct if and only if $r + n \leqq m$. The incorrectness for $r + n > m$ is shown as follows. We assume that *pop* is total: execution of *pop* with precondition $\mathbf{t} \leqq 0$ is allowed and gives i some address value. Consider a situation that r addresses are used and all processes have reserved an address by executing *pop*. Since $r + n > m$, the pigeonhole principle together with predicate (P0) implies that not all processes have reserved *different* addresses. So, there are different processes, say P and Q, that have reserved the same address i. Then the environment executes *Env*, so that the next action U preserves (P1). We then let process Q execute U and now process P violates (P0).

The correctness for $r + n \leqq m$ can be proved as follows. We write $i.Q$ to denote private variable i of process Q. We introduce a number of invariants. In all such predicates we universally quantify over addresses j and k and processes Q and T. The invariant (J0) is equivalent to (P0). The motivation for the other invariants is given in the proof of invariance. The invariants (J0), (J1) and (J2) express that the addresses found are not used, are all different and are not in the stack. Invariants (J3) and (J4) express that the addresses in the stack are not used and are all different. Invariant (J5) serves to determine the length of the stack. Predicate (J6) implies that the stack is nonempty when process Q can execute *pop*.

(J0) $$Q \textbf{ at } U \ \Rightarrow \neg \mathrm{used}.(i.Q),$$

(J1) $$Q, T \textbf{ at } U \wedge Q \neq T \ \Rightarrow i.Q \neq i.T,$$

(J2) $$Q \textbf{ at } U \wedge 0 \leqq k < \mathbf{t} \ \Rightarrow i.Q \neq \mathbf{x}.k,$$

(J3) $$0 \leqq k < \mathbf{t} \ \Rightarrow \neg \mathrm{used}.(\mathbf{x}.k),$$

(J4) $$0 \leqq j < k < \mathbf{t} \ \Rightarrow \mathbf{x}.j \neq \mathbf{x}.k,$$

(J5) $$(\#k :: \mathrm{used}.k) + (\#T :: T \textbf{ at } U) + \mathbf{t} = m,$$

(J6) $$\neg(Q \textbf{ at } U) \ \Rightarrow \mathbf{t} > 0.$$

The invariance is proved as follows. Predicate (J0) is preserved by *pop* because of (J3) and (J6). It is preserved by command U of other processes because of (J1). Predicate (J1) is preserved by *pop* because of (J2) and (J6). Predicate (J2) is preserved by *pop* because of (J4); it is preserved by *Env* because of (J0). Predicate (J3) is preserved by U because of (J2). It is also preserved by *Env*, as is easily verified. Predicate (J4) is preserved by *Env* because of (J3). Predicate (J5) is preserved by *pop*, U and *Env*, since in each case one summand increases with 1 and another summand decreases with 1. Predicate (J6) follows from invariant (J5), postulate (P1) and assumption $r + n \leqq m$. A straightforward verification shows that the predicates hold initially. The remaining details are left to the reader.

Notice that the actions *pop* and *push* must be atomic. For otherwise, invariance of (J0) cannot be guaranteed. We regard the atomic commands *pop* and *push* as too

coarse, however. For, both of them contain two shared variables (t and x) that can be modified by other processes.

It follows that we have to decide which atomic commands are allowed. We do not impose the condition, expressed in [12]: "Each expression E (in the program) may refer to at most one variable y that can be changed by another process while E is being evaluated, and E may refer to y at most once. A similar restriction holds for assignment statements $\mathbf{x} := E$". We only postulate that each atomic command S may refer to at most one variable y that can be changed by another process while S is being executed. So we allow multiple occurrences of one shared variable in an atomic command. Of course, our postulate is a weaker requirement for the commands and, hence, a stronger requirement for the implementation.

Actually, apart from private actions, we shall only use atomic commands of the following types

read: $u := \mathbf{x}$,

write: $\mathbf{x} := E$,

RS: $\langle u := \mathbf{x};\ \mathbf{x} := C \rangle$,

Con: $\langle$ **if** $\mathbf{x} = C$ **then** $\mathbf{x} := E;\ b := \mathit{false}$ **fi** $\rangle$,

where x is a shared variable, C is a constant, u and b are private variables and E is an expression in private variables. Tests of the form $\mathbf{x} = E$ are also used (they can be regarded as reading). Command *RS* is called a read-and-set action; command *Con* is called a consensus action.

Notice, that all atomic commands may be combined with arbitrary assignments to ghost variables.

3 A naïve solution

We now begin a development that leads to the other two solutions. Henceforth, the idea of a stack of free addresses is discarded and we use boolean variables to indicate whether an address is free.

We let the process names range over *process* as declared in

type $\mathit{process} = 0..n - 1$.

We introduce a boolean array free to indicate whether an address is free. It is declared by

var free: **array** *address* **of** *boolean*;

initially $(\forall k :: \mathsf{free}.k)$.

The environment uses F: $\mathsf{free}.k := \mathit{true}$ and, hence,

Env: $\langle \mathsf{free}.k := \mathit{true};\ \mathsf{used}.k := \mathit{false} \rangle$.

For the purpose of choosing an address, each process P may use the atomic read-and-set action

$$G\colon \langle b := \mathsf{free}.i;\ \mathsf{free}.i := \mathit{false}\rangle$$

where i and b are private variables of P.

It is clear that the actions *Env* of the environment preserve or re-establish the invariant

(P2) $$\neg\mathsf{free}.k \vee \neg\mathsf{used}.k.$$

Predicate (P2) also holds initially. We decide that the processes will also preserve (P2). Then $\mathsf{free}.k$ can be used to infer $\neg\mathsf{used}.k$. Since (P2) admits the possibility that both $\mathsf{free}.k$ and $\mathsf{used}.k$ are *false*, we introduce a ghost variable $\sigma.k$ to indicate the process that is the owner of address k. So we interpret $\sigma.k = Q$ to mean that address k has been found free and that Q is the only process that can use (or transfer) the address; $\sigma.k = \bot$ means that address k has no owner, and is free or used, according to the invariant

(P3) $$(\sigma.k = \bot) \equiv (\mathsf{free}.k \vee \mathsf{used}.k).$$

Ghost variable σ is declared by

var σ: **array** *address* **of** *process* $\cup \{\bot\}$;

initially $(\forall k :: \sigma.k = \bot)$.

It follows that (P3) holds initially. In view of the intention of σ, we extend command G of process P to

G: $\langle\ b := \mathsf{free}.i;\ \mathsf{free}.i := \mathit{false}$

; **if** b **then** $\sigma.i := P$ **fi** $\rangle$,

and we extend the setting command of $\mathsf{used}.i$ to

U: $\langle\ \mathsf{used}.i := \mathit{true};\ \sigma.i := \bot\ \rangle$.

Now it is easy to verify that commands G and U both preserve (P3). Moreover, since it is only executed under precondition $\mathsf{used}.k$, command *Env* also preserves (P3).

Whenever command G has postcondition b, process P has obtained a free address in i. In this way we arrive at the solution:

search0.P =

[**repeat**

choose $i \in$ *address*

; $\langle\ b := \mathsf{free}.i;\ \mathsf{free}.i := \mathit{false}$

; **if** b **then** $\sigma.i := P$ **fi** $\rangle$

until b

; U: $\langle\ \mathsf{used}.i := \mathit{true};\ \sigma.i := \bot\ \rangle$].

For the proof of partial correctness we also need the invariant

(P4) $$(\sigma.k = Q) \equiv (k = i.Q \wedge \mathit{fin}.Q),$$

where $\mathit{fin}.Q$ indicates that process Q is in the final stage of *search*: the repetition in *search* is about to terminate or Q is at U.

It is easy to see that (P4) holds initially and that it is preserved by the actions of process Q itself. It follows from (P3) and $Q \neq \bot$ that (P4) is preserved by the actions of all processes $P \neq Q$. Therefore (P4) is an invariant. The conjunction of (P3) and (P4) implies (P0). This proves partial correctness of *search0*.

Notice that we do not lose addresses, since the conjunction of (P2), (P3) and (P4) implies that

$$(\#k :: \mathsf{free}.k) + (\#k :: \mathsf{used}.k) + (\#Q :: \mathsf{fin}.Q) = m.$$

This equality also shows that, if $r + n < m$, there are always free addresses to be found.

The most obvious way to implement the choice of i is to modify a previous value of i by $i := i + 1$, say modulo m. Since other processes as well as the environment may have modified $\mathsf{free}.i$ between two inspections in command G, however, there is no easy way to guarantee that the repetition of *search0* terminates. In the next two sections we present two different solutions to this problem.

4 A solution in quadratic space

Inspired by Herlihy's algorithm for the wait-free implementation of an arbitrary data object in [9], we can solve the problem by partitioning the address space into n private pools, one for each process. For simplicity we assume that m is a multiple of n, say $m = c \cdot n$. Then *address* is the disjoint union of the sets $\mathit{addr}.P$ with $P \in \mathit{process}$ where $\mathit{addr}.P$ is given by

$$k \in \mathit{addr}.P \equiv c \cdot P \leqq k < c \cdot (P + 1).$$

We decide that each process fishes only in its own pool, i.e., the choice of i in *search* $0.P$ is restricted to the set $\mathit{addr}.P$. It follows that process P only sets $\mathsf{free}.k := \mathit{false}$ for $k \in \mathit{addr}.P$. This, again, implies that for $k \in \mathit{addr}.Q$ the predicate $\mathsf{free}.k$ is stable while process Q is in its searching repetition.

It also follows that we have the invariant

(P5) $$k \in \mathit{addr}.Q \;\Rightarrow\; (\mathsf{free}.k \vee \sigma.k = Q \vee \mathsf{used}.k).$$

Since, by (P1), the number of used addresses is bounded by r, we can now achieve a terminating search if $r < c$. For, then, when P is in its searching repetition, (P4) and (P5) imply that

$$(\forall k \in \mathit{addr}.P :: \mathsf{free}.k \vee \mathsf{used}.k).$$

Therefore, $r < c$ implies the existence of an address $k \in addr.P$ with free.k. This address remains free during the repetition. This shows that we may use

search1.P =
[$i := c \cdot (P + 1)$
; **repeat**
  $i := i - 1$
  ; $\langle\, b := \mathsf{free}.i;\ \mathsf{free}.i := false$
  ; **if** b **then** $\sigma.i := P$ **fi** $\rangle$
  until b
; $\langle\, \mathsf{used}.i := true;\ \sigma.i := \perp \rangle$].

With respect to space complexity, this solution requires $m = c \cdot n \geqq (r + 1) \cdot n$ addresses. Since r is often proportional to n,we regard it as a quadratic solution. The worst-case time complexity of *search1* is proportional to r.

Actually, in this case, since processes do not compete at the same address, there is a solution with simpler primitives:

search2.P =
[$i := c \cdot (P + 1)$
; **repeat** $i := i - 1$ **until** free.i
; $\langle \mathsf{free}.i := false;\ \sigma.i := P \rangle$
; $\langle \mathsf{used}.i := true;\ \sigma.i := \perp \rangle$].

This solution satisfies the condition of [12], see Sect. 2. It follows that our problem of finding free addresses does not require stronger primitives than read-write registers. So our problem has consensus number 1 in the sense of Herlihy [9].

5 A solution in linear space

We now develop a more flexible solution that requires less memory space. The idea is to avoid individual starvation by letting each searching process find addresses for all searching processes (this was inspired by the program in Fig. 14 of [9]). When some process has found a free address, it offers the address to its current private favorite. If the latter process is not searching, the process itself uses the address.

5.1 Program development

The starting point is *search0* of Sect. 3 with its invariants (P2) and (P3), but now a searching process may itself find a free address or it may obtain a free address from another process. Just as in Sect. 3, we use ghost variable $\sigma.k$ to indicate the process that owns address k. For the purpose of transferring addresses, we introduce a shared variable ad declared by

var ad: **array** *process* **of** *address* $\cup \{\perp\}$,

initially $(\forall Q :: \mathsf{ad}.Q \neq \perp)$.

We use $\mathbf{ad}.Q = \bot$ to indicate that process Q is searching and has not yet obtained a free address for its own use. We give command U the form

$$U\colon \langle\ \mathbf{used}.(\mathbf{ad}.P) := \mathit{true};\ \sigma.(\mathbf{ad}.P) := \bot \rangle.$$

Now specification (P0) is translated into

$$(\mathrm{Q0}) \qquad Q\ \mathbf{at}\ U\ \Rightarrow\ \mathbf{ad}.Q \neq \bot\ \wedge\ \neg\mathbf{used}.(\mathbf{ad}.Q).$$

Moreover, when Q is at U, other processes must not be able to modify $\mathbf{ad}.Q$. We shall therefore treat $\mathbf{ad}.Q$ as a consensus variable that can only be modified by processes $P \neq Q$ under precondition $\mathbf{ad}.Q = \bot$. Since $\mathbf{ad}.Q = \bot$ is an invitation for other processes to transfer an address to process Q, we postulate the invariant

$$(\mathrm{Q1}) \qquad \mathbf{ad}.Q = \bot\ \Rightarrow\ Q\ \mathbf{is}\ \mathit{searching}.$$

We give each process two private variables: i of type *address* and pf of type *process*. These variables are persistent or global in the sense that they keep their values while the process is not searching. They are initialized in an arbitrary way and they are modified by circular incrementation, i.e., by incrementation modulo m and n, respectively. The persistence of variable i is only used for the amortized complexity in Sect. 7. The persistence of pf is necessary for fairness and bounded delay.

If a process finds a free address by a successful execution of command G (see Sect. 3), it offers this address to its "private favorite", the process pf. The act of offering a free address to a process Q needs the precondition $\mathbf{ad}.Q = \bot$. Just as in Sect. 3, a private boolean variable b is used to indicate that address i has been found free and has not yet been transferred to some process. Therefore, array $\mathbf{ad}$ is modified by consensus actions of the form

$$H.Q\colon \langle\ \mathbf{if}\ \mathbf{ad}.Q = \bot\ \mathbf{then}\ \mathbf{ad}.Q := i;\ b := \mathit{false};\ \sigma.i := Q\ \mathbf{fi}\ \rangle.$$

In particular, command $H.pf$ is used to offer address i to the current private favorite pf. This action is preceded by a circular incrementation of pf. If command $H.pf$ has postcondition b, address i is still available for process P itself. Therefore, we then let P terminate its repetition and execute $H.P$ (in this case $\sigma.i$ need not be modified). If $H.P$ also has postcondition b, address i is still available but not needed. Then it is made free again and $\sigma.i$ is reset to $\bot$.

In this way, we arrive at procedure *search3* as given in Fig. 1. Notice that, since $\mathbf{ad}.P$ is only modified by the consensus action $H.P$, it follows from (Q0) that the expression $\mathbf{ad}.P$ is constant while command U is being executed, as was required in Sect. 1.

For the proof of correctness we need to argue about specific locations in the program in combination with the value of variable b. We therefore transform the while-program of Fig. 1 into the goto-program of Fig. 2. This goto program can be presented as a **do-od** loop with a variable pc explicitly appearing in the guards and updated at the end of each guarded command. Such a program would be in the spirit of Back's action systems, cf. [3], or of UNITY programs, cf. [7]. For the sake of brevity, however, we prefer to use goto statements.

Each number in the goto-program represents one atomic command. We have eliminated the boolean variable b and some goto statements. This is allowed since

```
search3.P =
[  ad.P := ⊥ ;  b := false
;  while (ad.P = ⊥) ∧ ¬b do
       i := (i + 1) mod m
   ;  ⟨  b := free.i ;  free.i := false
       ;  if b then σ.i := P fi  ⟩
   ;  if b then
          pf := (pf + 1) mod n
       ;  ⟨  if ad.pf = ⊥ then ad.pf := i ;  b := false ;  σ.i := pf fi ⟩
       fi
   od
;  if b then
       ⟨  if ad.P = ⊥ then ad.P := i ;  b := false fi ⟩
   ;  if b then ⟨ free.i := true ;  σ.i := ⊥ ⟩ fi
   fi
;  U :  ⟨ used.(ad.P) := true ;  σ.(ad.P) := ⊥ ⟩  ] .
```

Fig. 1. *search3* as a while-program

```
0    ad.P := ⊥
1    if ad.P ≠ ⊥ then goto 8 fi
2        i := (i + 1) mod m
3        if free.i then free.i := false ;  σ.i := P else goto 1 fi
4        pf := (pf + 1) mod n
5        if ad.pf = ⊥ then ad.pf := i ;  σ.i := pf ;  goto 1 fi
6    if ad.P = ⊥ then ad.P := i ;  goto 8 fi
7        free.i := true ;  σ.i := ⊥
8    used.(ad.P) := true ;  σ.(ad.P) := ⊥ ;  goto 0
```

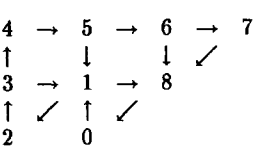

Fig. 2. Repeated *search3* as a goto-program

the program counter and b are private variables. In Fig. 2, we also give the directed graph of the possible ways for the flow of control. This graph is only an illustration, it is not used in the arguments. Since procedure *search* is contained in an unbounded repetition, command 8 of Fig. 2 contains **goto** 0.

5.2 Detailed invariants

The invariants (Q0) and (Q1) postulated above are not sufficiently detailed for the proof of correctness. So, *in addition to the invariants (P2) and (P3)*, we now postulate the following list of invariants. We write $[r..s)$ to denote the set of integers i with $r \leqq i < s$. We write pc to denote the program counter; this is a private variable of the acting process (which is usually called P). We write $pc.T$, $i.T$, etc., to denote the private variable pc, i, etc., of process T.

Predicate (R0) is a more explicit version of (Q1). Predicates (R1) and (R2) describe the regions where process Q holds responsibility of the addresses in $\mathrm{ad}.Q$

and $i.Q$. Predicate (R3) serves to show that the responsibilities for $i.Q$ and $\mathsf{ad}.Q$ do not interfere.

(R0) $$\mathsf{ad}.Q = \bot \;\Rightarrow\; pc.Q \in [1..7),$$

(R1) $$\mathsf{ad}.Q \neq \bot \;\wedge\; pc.Q \in [1..9) \;\Rightarrow\; \sigma.(\mathsf{ad}.Q) = Q,$$

(R2) $$pc.Q \in [4..8) \;\Rightarrow\; \sigma.(i.Q) = Q,$$

(R3) $$pc.Q \in [4..8) \;\Rightarrow\; i.Q \neq \mathsf{ad}.Q.$$

Before we turn to the proof of invariance, we first list a number of consequences of these predicates:

(C0) $$pc.Q = 8 \;\Rightarrow\; \neg\mathsf{free}.(\mathsf{ad}.Q) \wedge \neg\mathsf{used}.(\mathsf{ad}.Q),$$

(C1) $$pc.Q = 7 \;\Rightarrow\; \neg\mathsf{free}.(i.Q) \wedge \neg\mathsf{used}.(i.Q),$$

(C2) $$pc.P, pc.Q \in [4..8) \;\wedge\; i.P = i.Q \;\Rightarrow\; P = Q,$$

(C3) $$pc.P, pc.Q \in [1..9) \;\wedge\; \mathsf{ad}.P = \mathsf{ad}.Q \neq \bot \;\Rightarrow\; P = Q,$$

(C4) $$pc.P \in [4..8) \;\wedge\; pc.Q \in [1..9) \;\Rightarrow\; i.P \neq \mathsf{ad}.Q.$$

Since $Q \neq \bot$, predicate (C0) follows from (R0), (R1) and (P3). Similarly, (C1) follows from (R2) and (P3). Predicate (C2) follows from (R2). Predicate (C3) follows from (R1). Predicate (C4) follows from (R1)–(R3). Notice that the specification (Q0) follows from (R0) and (C0).

5.3 The proof of safety

We turn to the proof of invariance of (P2), (P3) and (R0) through (R3). Predicates (P2) and (P3) hold initially. Since initially $\mathsf{ad}.Q \neq \bot$ and $pc.Q = 0$ for all Q, the predicates (R0)–(R3) hold initially, because of falsity of the antecedent.

The remainder of the proof of invariance is not more than a huge but elementary case analysis. We have checked the following proof by means of the theorem prover NQTHM of Boyer and Moore, cf. [6], in the same way as reported in [11].

We first deal with the action *Env* as given in Section 3. Since *Env* only modifies free and used, we notice that used and free only occur in (P2) and (P3). Just as in Sect. 3, it is easy to see that the action *Env* preserves (P2) and (P3). It follows that we need not consider action *Env* anymore. So we only consider the actions of the processes. We now discuss the predicates (P2), (P3) and (R0) through (R3), consecutively.

Predicate (P2) is threatened only by commands 7 and 8. By (C1), command 7 of process P has precondition $\neg\mathsf{used}.(i.P)$ and, hence, preserves (P2). Similarly, by (C0), command 8 has precondition $\neg\mathsf{free}.(\mathsf{ad}.P)$ and, hence, preserves (P2).

Predicate (P3) is threatened only by commands 3, 5, 7 and 8. It is preserved by command 3 because of (P2). It is preserved by command 5 of P because (R2) with $Q := P$. Predicate (P3) is clearly preserved (established) by commands 7 and 8.

Predicate (R0) is threatened only by commands 0, 1 and 6 of process Q. It is preserved by command 0, since there $pc.Q$ becomes 1. It is preserved by 1 because

of the test $\mathbf{ad}.P \neq \perp$. It is preserved by 6 since command 6 has postcondition $\mathbf{ad}.P \neq \perp$.

Predicate (R1) is threatened by commands 0, 3, 5, 6, 7 and 8. Command 0 preserves (R1) since the antecedent is made *false*. Command 3 of process P preserves (R1) because of (P3) with $k := i.P$. Command 5 of P establishes (R1) for $Q = pf.P$. Predicate (R1) for $Q \neq pf.P$ is preserved because of (C4) with $Q := P$ and $T := Q$. Command 6 of Q preserves (R1) because of (R2). Command 7 preserves (R1) because of (C4) with $Q := P$ and $T := Q$. Command 8 of process P preserves (R1) because of (C3).

Predicate (R2) is threatened only by the commands 3, 5, 7 and 8. Command 3 of Q preserves (R2) since it makes $pc.Q = 1$ or establishes the consequent. For $P \neq Q$, command 3 of P preserves (R2) because of (P3) with $k = i.P$. Commands 5 and 7 of Q preserve (R2) of Q because of the modification of $pc.Q$. Commands 5 and 7 of $P \neq Q$ preserve (R2) of Q because of (C2). Command 8 preserves (R2) because of (C4).

Predicate (R3) is threatened only by commands 3, 5 and 6. Command 3 of P threatens (R3) only if $P = Q$ and $\mathbf{free}.i$; then (P3) implies $\sigma.i = \perp$ so that (R1) implies $i.P \neq \mathbf{ad}.P$. Therefore, (R3) is preserved. Command 5 only threatens (R3) if $\mathbf{ad}.pf = \perp$ and $P \neq Q = pf$ and $pc.Q \in [4..8)$; but then (C2) implies $i.P \neq i.Q$. Again (R3) is preserved. Command 6 of P preserves (R3) for $P \neq Q$ since it does not modify $pc.Q, i.Q$ or $\mathbf{ad}.Q$. It preserves (R3) for $P = Q$ since $pc.P$ becomes 8 if $\mathbf{ad}.P$ changes.

This concludes the proof of invariance of (P2), (P3) and (R0)–(R3), and thus the proof of partial correctness.

5.4 No dangling addresses

For the purpose of progress, we first show that the algorithm does not produce dangling addresses. More precisely, we show the invariance of

$$\text{(R4)} \qquad \sigma.k \neq \perp \;\Rightarrow (k = \mathbf{ad}.(\sigma.k) \wedge pc.(\sigma.k) \in [1..9)) \vee (k = i.(\sigma.k) \wedge pc.(\sigma.k) \in [4..8)).$$

Predicate (R4) is threatened only by the commands 0, 3, 5, 6, 7 and 8. Command 0 can only falsify the consequent of (R4) if $\sigma.k = P$ but then the consequent is *false* already, since $pc.P = 0$. Command 3 is relevant only if $\mathbf{free}.i$ and $k = i$, but then it establishes the consequent of (R4). Command 5 is only relevant if $\mathbf{ad}.pf = \perp$. For $k = i$, one uses (R0) with $Q := pf$ to conclude that the first disjunct of the consequent of (R4) is established. If $k \neq i$ then $\sigma.k$ does not change; for $\sigma.k = P$ the second disjunct of the consequent is *false*, for $\sigma.k \neq P$ the second disjunct of the consequent does not change; so predicate (R4) is threatened only when $\mathbf{ad}.(\sigma.k) = k$ holds and is made *false*; then the precondition has $\mathbf{ad}.(\sigma.k) = k \neq \perp$ so that $\sigma.k \neq pf$. Therefore, (R4) is preserved. Command 6 does not modify $\sigma.k$. It only threatens the consequent of (R4) if $P = \sigma.k$ and $k = i$, but in that case the first disjunct of the consequent becomes *true*. Command 7 can only falsify the second disjunct of the consequent of (R4), but in that case it also falsifies the antecedent. It therefore preserves (R4). Command 8 can only falsify the consequent by making

$pc.P = 0$, but then $P = \sigma.k$ and $k = \mathbf{ad}.(\sigma.k)$, so that the antecedent is made *false*. This proves the invariance of (R4).

It may well be that the algorithm is to be applied continuously in an environment where, over time, the number of processes may change slowly. In that case, the number n of declared processes need not be the number of actual processes but is only an upper bound. The bounds in the time complexity may depend both on the constant n that occurs in the algorithm and on the number of actual processes. We therefore assume a constant set V to be given that contains the actual processes. So V is a subset of *process* and all processes $Q \notin V$ are regarded as always idle, in the sense that $\mathbf{ad}.Q \neq \bot$ and $pc.Q = 0$. Of course, processes $Q \in V$ can also be idle, temporarily or permanently. We use the constant v to denote the number of elements of V. It follows from (P3) and (R4) that we have

$$\mathbf{free}.k \vee \mathbf{used}.k \vee (\exists T \in V :: k = i.T \vee k = \mathbf{ad}.T).$$

Using (P1) we now obtain

(C5) $$(\#k :: \neg \mathbf{free}.k) \leqq r_1, \text{ where } r_1 = r + 2 \cdot v.$$

We can now state the weakest possible progress assumption for *search3*. It is clear that process $P \in V$ can never find a free address if all processes $Q \in V \setminus \{P\}$ have stopped executing in a state where

$$(\forall k :: \neg \mathbf{free}.k \wedge \neg \mathbf{used}.k \wedge (\exists T \in V \setminus \{P\} :: k = i.T \vee k = \mathbf{ad}.T)).$$

This shows that starvation is possible if $m \leqq 2 \cdot v - 2$.

Let us therefore assume that $2 \cdot v - 1 \leqq m$ and that the environment guarantees that, whenever some addresses are used, eventually at least one address is made free. The latter condition can be expressed in the temporal formula

$$\Box((\exists k :: \mathbf{used}.k) \Rightarrow \Diamond(\exists k :: \mathbf{free}.k)).$$

Under these assumptions individual starvation does not occur. In fact, assume that process P is searching in the sense that it did not stop executing and $\mathbf{ad}.P = \bot$. During this search, no other process can obtain a free address more than n times. Since at most $2 \cdot (v - 1)$ addresses are kept by processes that have stopped executing, always at least one address is used or free or kept by some active process. The temporal formula therefore implies that free addresses remain being produced. Therefore, eventually, P or some other process will get a free address for process P. The details of this proof of fairness are left to the reader, since we want to emphasize bounded delay and amortized time complexity.

6 Bounded delay

Since we cannot hope for bounded delay if it is possible that all addresses k have $\neg \mathbf{free}.k$, we now postulate that r_1 introduced in (C5) satisfies $r_1 < m$.

In order to prove bounded delay for process $Q \in V$, we assume that process Q executes command 0. It now suffices to prove that $\mathbf{ad}.Q \neq \bot$ holds after a bounded number of actions of Q. For this purpose, we introduce an expression $dis(P, Q)$ for *the number of addresses that process P must find free in command*

3 before offering an address to process Q in command 5. The expression is defined by

$$dis(P,Q) =$$
$$(Q - pf.P) \bmod n - \#(pc.P = 4)$$
$$+ n \cdot \#(Q = pf.P \wedge pc.P \neq 5),$$

where we use $\#A$ to denote 1 if A holds, and 0 otherwise (the operator **mod** we use satisfies $0 \leqq x \bmod n < n$ for all integers x, possibly negative).

We proceed to show that $dis(P,Q)$ satisfies its informal specification. We first prove that $0 \leqq dis(P,Q) \leqq n$ always holds. In fact, $dis(P,Q) < 0$ implies $Q = pf.P$ and $pc.P = 4$, but then $dis(P,Q) = n - 1$. On the other hand, $dis(P,Q) > n$ also leads to a contradiction. We next investigate how $dis(P,Q)$ changes under the actions of the processes. Clearly, $dis(P,Q)$ can only change when P executes one of the commands 3, 4 or 5. Actually, $dis(P,Q)$ does not change if P executes command 4. In fact, if P executes command 4 with $pf.P \neq Q$, it decrements both operands of the subtraction without modifying the final summand. So $dis(P,Q)$ is not changed. If process P executes command 4 with $pf.P = Q$, the subtraction changes from -1 to $n - 1$ and the final summand changes from n to 0. So, again, $dis(P,Q)$ is not changed. Command 5 of P only modifies $dis(P,Q)$ if $pf.P = Q$. In that case, $dis(P,Q) = 0$ and process P offers a free address to Q. So, the postcondition $\mathsf{ad}.Q \neq \bot$ is satisfied. Command 3 of P only modifies $dis(P,Q)$ if $\mathsf{free}.(i.P)$ holds. In that case, process P finds a new address and $dis(P,Q)$ is decremented with 1. This shows that $dis(P,Q)$ is indeed the number of addresses that process P must find free in command 3 before offering an address to process Q in command 5.

We now define the "waiting function" $wa.Q$ by

$$wa.Q = (\sum T \in V :: dis(T,Q)).$$

It follows from the above arguments that $wa.Q$ satisfies the following lemma.

Lemma 0. (a) $0 \leqq wa.Q \leqq v \cdot n$.
(b) *If some process increases* $wa.Q$, *the postcondition satisfies* $\mathsf{ad}.Q \neq \bot$.
(c) *If some process in command 3 finds a new address,* $wa.Q$ *decreases with 1.*

So now we have to guarantee that sufficiently many free addresses are found in command 3. Since process P searches through the range *address* in a circular way, we define an expression $wf.k$ for the amount of not-free addresses j in front of k. The addresses gets weights that decrease linearly according to the number of steps needed to reach the address from k. Writing $\langle x \rangle = x \bmod m$ for any integer x, we define

$$wf.k = (\sum j: \neg \mathsf{free}.j: \langle k - j \rangle)$$

where j ranges over addresses. Notice that address $j = \langle k + 1 \rangle$ has the maximal weight $\langle -1 \rangle = m - 1$ and that address $j = k$ has the minimal weight $\langle 0 \rangle = 0$. Function $\langle - \rangle$ satisfies

$$\langle \langle x \rangle \pm \langle y \rangle \rangle = \langle x \pm y \rangle,$$
$$\langle x + 1 \rangle - \langle x \rangle = 1 - m \cdot \#(\langle x + 1 \rangle = 0).$$

Since the reference point k can be incremented modulo m, it is useful to estimate the difference

$$
\begin{aligned}
& wf.\langle k+1\rangle - wf.k \\
= \; & \{\text{distributivity and above rule}\} \\
& (\sum j: \neg\mathsf{free}.j: \langle k+1-j\rangle - \langle k-j\rangle) \\
= \; & \{\text{above rule}\} \\
& (\sum j: \neg\mathsf{free}.j: 1 - m\cdot \#(\langle k+1-j\rangle = 0)) \\
= \; & \{\text{counting, one-point rule}\} \\
& (\#j::\neg\mathsf{free}.j) - m\cdot \#(\neg\mathsf{free}.\langle k+1\rangle) \\
\leqq \; & \{\ (\text{C5})\ \} \\
& r_1 - m\cdot \#(\neg\mathsf{free}.\langle k+1\rangle).
\end{aligned}
$$

Since we have $r_1 < m$ by assumption, this shows that the circular incrementation of the reference point k can only increase $wf.k$ if the new address $\langle k+1\rangle$ is free. Since the act of finding a new address is contained in command 3, we define the reference point for process Q in such a way that it has a circular incrementation in command 3 with $i.Q$ as its new value. The reference point is defined by

$$ii.Q = \textbf{if } pc.Q \neq 3 \textbf{ then } i.Q \textbf{ else } \langle i.Q - 1\rangle \textbf{ fi}.$$

Indeed, one can easily verify that $ii.Q$ only changes when process Q executes command 3 and that command 3 of Q has the effect of $ii.Q := ii.Q + 1$. We now define the weight function for process Q by

$$wfi.Q = wf.(ii.Q).$$

Its main properties are contained in

Lemma 1. (a) *$wfi.Q$ decreases with at least $m - r_1$ when process Q executes command 3 with precondition* $\neg\mathsf{free}.(i.Q)$.
(b) *$wfi.Q$ can only increase when some process P executes command 3 with the precondition* $\mathsf{free}.(i.P)$. *The increase is at most r_1 if $P = Q$. It is at most $m - 1$ if $P \neq Q$.*
(c) $0 \leqq wfi.Q \leqq r_1 \cdot m$.

Proof. (a) This follows from the above calculation.

(b) If process Q executes command 3 with precondition $\mathsf{free}.(i.Q)$, the new not-free address gives contribution 0 to the new value of $wfi.Q$. Therefore, the above calculation implies that $wfi.Q$ increases with at most r_1. It is clear that other actions of Q do not increase $wfi.Q$. It is also clear that a process $P \neq Q$ can only increase $wfi.Q$ by executing command 3 with precondition $\mathsf{free}.(i.P)$ and that the increase is at most $m - 1$.

(c) This follows from (C5) and the definitions of wfi and wf. □

We now combine function $wfi.Q$ with the function $wa.Q$ considered earlier. If process Q executes command 3 with precondition $\mathsf{free}.(i.Q)$, it increments $wfi.Q$

with at most r_1 and it decrements $wa.Q$ with 1. If some other process, say $P \in V$ with $P \neq Q$, increments $wfi.Q$, it does so by setting $\mathbf{free}.(i.P)$ to *false*. Then $wfi.Q$ increases with at most $m - 1$ and function $wa.Q$ decreases with 1. This implies that

$$vf.Q = wfi.Q + m \cdot wa.Q$$

only increases when variable $\mathbf{ad}.Q \neq \perp$ is being established. Whenever process Q executes command 3, the value of $vf.Q$ decreases with at least $m - r_1$. This implies that $vf.Q$ decreases with $t \cdot (m - r_1)$ when Q executes its loop body t times while $\mathbf{ad}.Q = \perp$ remains valid.

It follows from the bounds for wfi and wa that $0 \leqq vf.Q \leqq m \cdot (r_1 + v \cdot n)$. This implies that, if Q executes its loop body t times while $\mathbf{ad}.Q = \perp$ remains valid, then $t \leqq C$ where C is given by

$$C = m \cdot (r_1 + v \cdot n) \; \mathbf{div}(m - r_1).$$

Consequently, during one call of *search3*, process Q executes its loop body at most $C + 1$ times. This proves that *search3*.Q terminates within a bounded number of actions of process Q.

Complexity. In order to estimate the complexity of *search3*, we assume that $m = e \cdot r_1$ for some fixed rational number $e > 1$. We assume that r and v (and hence r_1) are linear in n. Then the space complexity $m = e \cdot r_1$ is linear in n.

With respect to the time complexity, the time bound for *search3*.Q is proportional to $C + 1 \approx [e/(e - 1)]\,(r_1 + v \cdot n)$, which is quadratic in n. If one would take $m = r_1 + d$ for some natural number $d > 0$, then $C + 1$ and the time bound for *search3*.Q would be cubic in n, but it would not get worse, even if $d = 1$.

If we have to choose m for a fixed value of r_1, we notice that the time complexity is proportional to $e/(e - 1)$ and the space complexity proportional to e. It seems reasonable to choose m such that the product of these constants is minimal. A small calculation shows that this implies $e = 2$.

Remark. Alternatively, the introduction of $ii.Q$ can be avoided if one is willing to regard commands 2 and 3 as a single atomic command. This can be justified by means of the atomicity rule 6.26 of [1].

7 Amortized complexity of the system

In this section, we show that the amortized complexity of *search3* for the combined system of processes is linear in the number v of actual processes. Amortized complexity is defined here as the order of an upper bound of the number of atomic actions of all processes divided by the number of addresses found, during long execution sequences.

For this purpose, we introduce a ghost variable β to count the number of addresses found, more precisely, the number of times *search3* has been completed. So β is incremented with 1 whenever some process executes command 8. We introduce a ghost variable α to measure the activity of the processes, more precisely to count the number of times command 3 has been executed. So α is incremented with 1 whenever some process executes command 3. It can be shown that the total number of atomic actions of the processes is bounded by $5 \cdot \alpha + 4 \cdot \beta + 3$. Therefore,

an upper bound of the asymptotic ratio α/β is a measure for the amortized complexity.

We shall prove that $\alpha \leqq A \cdot \beta + B$ for certain constants A and B and that A is proportional to v. Actually, we shall prove a stronger inequality, namely that, if at some moment $\alpha = \alpha_0$ and $\beta = \beta_0$, then henceforth $\alpha - \alpha_0 \leqq A \cdot (\beta - \beta_0) + B$. The above arguments show that A is a measure for the amortized complexity.

Remark. The second inequality is indeed stronger than the first one. In fact, the first inequality allows, for every number K, that β is constant while α is incremented K times. The second inequality however yields the bound $K \leqq B$.

We first introduce a potential function for the slack of β:

$$pot = (\#\, T \in V :: pc.T \in [4..9)) + (\#\, T \in V :: pc.T \in [1..9) \wedge \mathsf{ad}.T \neq \bot).$$

Function *pot* increases with 1 whenever some process finds a free address in command 3, since then the first summand is incremented. Function *pot* does not change when a process executes the then-part of command 5, since then the first summand is decremented and the second summand is incremented because of invariant (R0). It is easy to see that command 6 does not decrement *pot*. Command 8 decrements *pot* with 2 because of (R0). The other commands leave *pot* unchanged. It now follows that $2 \cdot \beta + pot$ never decreases and that it increases whenever some process finds a new address in command 3.

We use the weight function *wfi* of Section 6 to define a global weight function

$$gw = (\sum T \in V :: wfi.T).$$

It follows from Lemma 1 that, whenever some process $P \in V$ executes command 3 with precondition $\neg \mathsf{free}.(i.P)$, it decrements *gw* with at least $m - r_1$. Whenever some process P finds a free address in command 3, it may increment $wfi.P$ with at most r_1 and it may increment the values $wfi.T$ for $T \in V \setminus \{P\}$ with less than m. So it increments *gw* with at most $r_1 + (v-1) \cdot m$. Every execution of command 3 increments α with 1. It follows that $(m - r_1) \cdot \alpha + gw$ only increases when some process finds a free address in command 3 and that it then increases with at most $v \cdot m$.

We now combine the assertions concerning $2 \cdot \beta + pot$ and $(m - r_1) \cdot \alpha + gw$ and obtain the result that the integer

$$v \cdot m \cdot (2 \cdot \beta + pot) - ((m - r_1) \cdot \alpha + gw)$$

never decreases. If we now begin a sequence of observations in a state where $\alpha = \alpha_0$ and $\beta = \beta_0$, etc., then henceforth we have the invariant

$$(m - r_1) \cdot (\alpha - \alpha_0) + (gw - gw_0) \leqq v \cdot m \cdot (2 \cdot \beta - 2 \cdot \beta_0 + pot - pot_0).$$

Since $pot - pot_0 \leqq 2 \cdot v$ and $gw_0 - gw \leqq v \cdot m \cdot r_1$, we obtain

$$\alpha - \alpha_0 \leqq A \cdot (\beta - \beta_0 + v + \tfrac{1}{2} \cdot r_1),$$

where $A = 2 \cdot v \cdot m/(m - r_1)$. This is indeed equivalent to an inequality of the form as announced. If we take $m = e \cdot r_1$ as before, we get the amortized time complexity $A = 2 \cdot v \cdot e/(e - 1)$, which indeed is linear in v.

8 Average behaviour

In this section we present a modification of *search3*, for which the results of the Sects. 5–7 remain valid, but which has a better average time complexity.

In fact, the average behaviour of *search3* suffers from the clustering of the values i of different processes. It is difficult to make a quantitative stochastic analysis, but the qualitative explanation is as follows. As soon as values i are clustered together, the processes with the rightmost values i find more free addresses so that their values i move more slowly. It follows that such clusters tend to be preserved. This has the effect that the processes search more or less in the same area and therefore find less free addresses than might be expected.

8.1 A minor modification

The remedy is to give each process its own search strategy. So we replace the circular incrementation in command 2 of Fig. 2 by $i := (i + u.P)$ **mod** m, where each process P has its own constant $u.P$. In order to preserve the results of the Sects. 6 and 7, we impose the condition that $u.P$ have greatest common divisor 1 with m. In fact, this implies the existence of a unique address $w.P$ such that the product satisfies $\langle w.P \cdot u.P \rangle = 1$. We replace the definition of ii by

$$ii.Q = \textbf{if } pc.Q \neq 3 \textbf{ then } i.Q \textbf{ else } \langle i.Q - u.Q \rangle \textbf{ fi}.$$

One can verify that $ii.Q$ is modified only when process Q executes command 3 and that it then is incremented with $u.Q$ modulo m. We redefine wfi by

$$wfi.Q = (\sum j: \neg \mathsf{free}.j: \langle w.Q \cdot (ii.Q - j) \rangle),$$

and observe that

$$\langle w.Q \cdot (k + u.Q - j) \rangle = \langle w.Q \cdot (k - j) + 1 \rangle.$$

It then turns out that Lemma 1 remains valid. Therefore, with these modifications the results of the Sects. 6 and 7 remain valid.

It remains to choose the constants $u.Q$. The choice $u.Q = 1$ for all Q is just the old solution. The best average behaviour is to be expected if all constants $u.Q$ are different. So, one can just take u to be an enumeration of addresses relatively prime to m.

If one wants a homogeneous solution, one can choose n and m in such a way that one can take the elements $u.P$ to form a group of invertible elements in the ring of integers modulo m. Then all processes are treated in the same way, since multiplication of addresses with $u.Q$ permutes the rôles of all processes and gives process Q the rôle of the process P with $u.P = 1$. We see no reason, however, to assume that other distributions of u lead to unfair treatment of some of the processes.

$r =$	5		8		11		14		16	
	# 8	# 2	# 8	# 2	# 8	# 2	# 8	# 2	# 8	# 2
0:	3	29	2	27	3	44	2	47	1	63
1:	8	22	5	25	8	42	7	43	6	61
2:	8	29	6	26	7	41	5	43	5	58
3:	6	22	4	24	4	44	3	46	2	61
4:	6	27	4	27	4	43	2	49	2	62
5:	5	28	5	26	3	43	2	45	1	59
#8/#2	.23 (.72)		.17 (.56)		.11 (.39)		.077 (.22)		.047 (.11)	

Fig. 3. Results with $m = 18$, $n = 6$ and $u.Q = 1$ for all Q

$r =$	5		8		11		14		16	
	# 8	# 2	# 8	# 2	# 8	# 2	# 8	# 2	# 8	# 2
0:	12	31	8	28	6	30	5	47	3	66
1:	17	24	11	25	8	26	10	41	6	62
2:	13	26	11	25	10	24	7	42	7	58
3:	16	25	7	28	9	25	7	43	5	59
4:	14	29	11	26	8	27	5	47	5	61
5:	15	23	10	27	6	29	6	41	6	54
#8/#2	.55 (.72)		.36 (.56)		.29 (.39)		.15 (.22)		.089 (.11)	

Fig. 4. Results with $m = 18$, $n = 6$ and $u.Q = 1, 5, 7, 11, 13, 17$

8.2 Results of simulations

We have performed some simulations to test the effect of this modification on the average behaviour. In these simulations we forced the environment to preserve the quality $(\#k :: \mathsf{used}.k) = r$. A random number generator was used to choose the next acting process and, for the environment, to choose the address to be given free. The interesting factor is the ratio between the number of actions 8 and the number of actions 2 (or 3). This ratio may be regarded as the productivity of the search and it can be compared with the maximal productivity $(m - r)/m$. In the Figs. 3 and 4 we represent the results of ten test sequences for a system of $n = 6$ processes and $m = 18$ addresses. Each double column represents one test sequence. The first row gives the number r of used addresses. The second row gives headers to distinguish the columns for commands 8 and 2. The next six rows give, for each of the six processes, the numbers ($\#8$) of executions of command 8 and the numbers ($\#2$) of executions of command 2. In the bottom row, we give the ratio $\#8/\#2$, where both quantities are summed over all processes; for comparison this ratio is followed by the fraction $(m - r)/m$ between parentheses. Figure 3 contains the results for *search3* as given in Fig. 2. Figure 4 contains the results for the modification presented above. In this case we use the six values $1, 5, 7, 11, 13, 17$ for $u.Q$. Comparison between the Figs. 3 and 4 shows that here the average performance of the modification is twice as good as the original version.

It turns out that, even in cases where the algorithm is not guaranteed to be wait-free, the average performance seems to be quite good. In fact, in the cases of the two tables the algorithm is only guaranteed to be wait-free for $r \leqq 5$. One may

also notice that the first process is, generally, the least productive one. This is a consequence of our choice to give short test sequences and to let each test sequence begin with $pf.Q = 0$ for all Q. In that way, process 0 is an ill-favoured process.

9 Comparison

The solution in quadratic space of Section 4 is simpler and has better worst-case time complexity than the one of Section 5. It is therefore to be preferred in cases where memory is no problem and in real time applications. It may also be preferred because of its use of simpler primitives, especially in the form of *search2*.

In some applications space complexity is a bottleneck. For instance, in [9] and [10], the addresses to be found are indices of an array of length m with elements of type X, where the values of X may be states of a data base. In that case, quadratic complexity of m may be prohibitive and the solution in linear space of Sect. 5, as modified in Sect. 8, may be preferred.

We assume that, in most applications, the performance of the latter solution is acceptable. Indeed, the worst-case time complexity may be quadratic, but the amortized complexity is linear and the average productivity seems to be a fair approximation of the maximal productivity that can be expected. The simulations show that the solution has graceful degradation: it is also applicable (though no longer wait-free) if almost all addresses are used.

We do not know other fine-grain solutions of the problem that have been proved completely. It seems to be easy to construct hybrid solutions in between our second and third solution, but we do not expect better behaviour from such a solution than from our third solution with the same address space. We have considered an alternative of our third solution in which the private favorites $pf.P$ are replaced by one common favorite. We rejected that solution, however, since it only introduced more proof obligations and more communication overhead, while having the same quadratic worst-case complexity.

References

1. Apt, K.R., Olderog, E.-R.: Verification of sequential and concurrent programs. New York: Springer 1991
2. Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., Reischuk, R: Renaming in an asynchronous environment. J. ACM **37** (1990) 524–548
3. Back, R.J.R., Sere, K.: Stepwise refinement of action systems. In: J.L.A. van de Snepscheut (ed.): Mathematics of Program Construction (Lect. Notes Comput. Sci., vol. 375, pp. 115–138) Berlin, Springer: 1989
4. Bar-Noy A., Dolev, D.: Shared-memory vs. message-passing in an asynchronous distributed environment. In Proc. 8th ACM Symp. on principles of distributed computing, pp. 307–318, 1989
5. Borowsky, E., Gafni, A.: Immediate snapshots and fast renaming. In: Proc. 12th ACM Symp. on principles of distributed computing, pp. 41–52, 1993
6. Boyer, R.S., Moore, J.: A computational logic handbook. Boston: Academic Press 1988
7. Chandy, K.M., Misra, J.: Parallel program design, a foundation. Reading, MA: Addison-Wesley 1988
8. Dijkstra, E.W.: A class of allocation strategies inducing bounded delays only. Tech. Rept., Tech. Univ. Eindhoven, EWD 319, 1971

9. Herlihy, M.P., Wait-free synchronization. ACM Trans. Program. Languages Systems. **13** (1991) 124–149
10. Hesselink, W.H.: Wait-free linearization with an assertional proof. Distributed Computing **8** (1994) 65–80
11. Hesselink, W.H.: Wait-free linearization with an mechanical proof. Computing Science Notes Groningen CS 9306, to appear in Distributed Computing
12. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs. Acta Inform. **6** (1976) 319–340